مراجعة للمكدس والرتل واللائحة

نحتاج في الخوارزميات إلى استخدام تقنيات برمجية وأنواع معطيات كالمكدس Stack والرتل Queue واللائحة List.

يمكن برمجة هذه التقنيات من الصفر، أو الاعتماد على قوالب مكتبات جاهزة (:Standard Template Library STL) تقدمها لغة ++C للتسهيل والمرونة.

في الأمثلة الثلاثة التالية اعتمدنا على تقديم هذه التقنيات بواسطة المكتبات الجاهزة: #include <stack> و #include <queue> و #include <list>

مثال على طريقة عمل المكدس بواسطة مكتبة جاهزة هي #include <stack>

سنستخدم التوابع الجاهزة التالية:

push() يقوم بإدخال ودفع قيمة للمكدس

pop() يقوم بإخراج وسحب قيمة من المكدس

empty() يفحص فيما إذا كان المكدس فارغاً أم لا

top() يعيد القيمة الموجودة في رأس المكدس

size() يعيد عدد عناصر المكدس

```
#include <iostream>
#include <stack>
using namespace std;
void showstack(stack <int> s)
{
	while (!s.empty())
	{
		cout << '\t' << s.top();
		s.pop();
	}
	cout << '\n';
}
int main ()
{
	stack <int> s;
	s.push(10);
	s.push(30);
```



```
    s.push(20);
    s.push(5);
    s.push(1);
    cout << "The stack is : ";
    showstack(s);
    cout << "\ns.size() : " << s.size();
    cout << "\ns.top() : " << s.top();
    cout << "\ns.pop() : ";
    s.pop();
    showstack(s);
    system ("pause");
    return 0;
}
```

الخرج

```
The stack is :     1     5     20     30     10

s.size() : 5
s.top() : 1
s.pop() :     5     20     30     10
```

مثال على طريقة عمل الرتل بواسطة مكتبة جاهزة هي `#include <queue>`

سنستخدم التوابع الجاهزة التالية:

push() يقوم بإدخال قيمة للرتل

pop() يقوم بإخراج وسحب قيمة من الرتل

empty() يفحص فيما إذا كان الرتل فارغاً أم لا

front() يعيد القيمة الموجودة في أول الرتل

back() يعيد القيمة الموجودة في آخر الرتل

size() يعيد عدد عناصر الرتل

```
#include <iostream>
#include <queue>
using namespace std;
void showq(queue <int> gq)
{
    while (!gq.empty())
    {
        cout << '\t' << gq.front();
        gq.pop();
    }
    cout << '\n';
}
```

```cpp
int main()
{
    queue <int> gquiz;
    gquiz.push(10);
    gquiz.push(20);
    gquiz.push(30);

    cout << "The queue gquiz is : ";
    showq(gquiz);

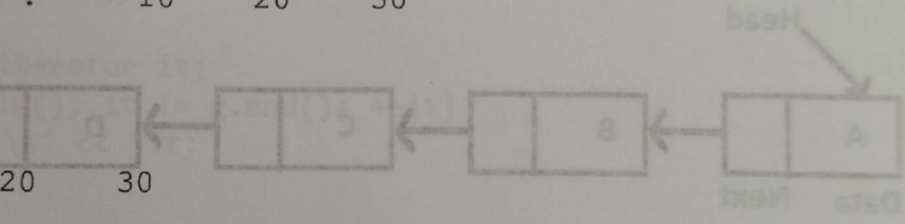
    cout << "\ngquiz.size() : " << gquiz.size();
    cout << "\ngquiz.front() : " << gquiz.front();
    cout << "\ngquiz.back() : " << gquiz.back();

    cout << "\ngquiz.pop() : ";
    gquiz.pop();
    showq(gquiz);
     system("pause");
    return 0;
}
```

الخرج:

```
The queue gquiz is :    10    20    30

gquiz.size() : 3
gquiz.front() : 10
gquiz.back() : 30
gquiz.pop() :     20    30
```



اللوائح المترابطة Linked List

قبل الحديث عن اللوائح يجب الحديث عن المصفوفات:

المصفوفة التقليدية: وهي ما كنا نتعامل معه في مقرر خوارزميات1. لا يمكن تحديد حجم هذه المصفوفة أثناء زمن التنفيذ.

المصفوفة الديناميكية: تعتمد على المؤشرات ويمكن تحديد حجم المصفوفة أثناء زمن التنفيذ، إلا أنها لا تدعم استخدام توابع جاهزة وبالتالي سيكون التعامل معها غير سهل.

Vector: هو نمط معطيات يشبه المصفوفة الديناميكية السابقة وموجود في مكتبة جاهزة وبالتالي يدعم استخدام توابع جاهزة لتسهيل التعامل معه.

محاسن المصفوفات والنمط الجاهز Vector: سهولة التجوال والوصول العشوائي داخل المصفوفة بواسطة دليل المصفوفة.

مساوئ المصفوفات والنمط الجاهز Vector: عملية حذف عنصر أو إضافة عنصر تتطلب معالجة كبيرة وعمليات ازاحة.

يوضح الشكل التالي عناصر مصفوفة عددها 9 عناصر.

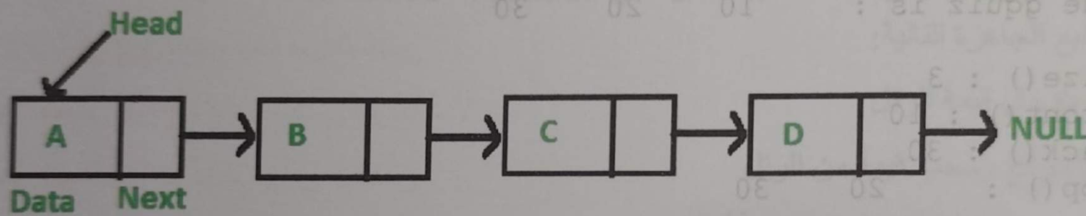
40	55	63	17	22	68	89	97	89	
0	1	2	3	4	5	6	7	8	<- Array Indices

Array Length = 9
First Index = 0
Last Index = 8

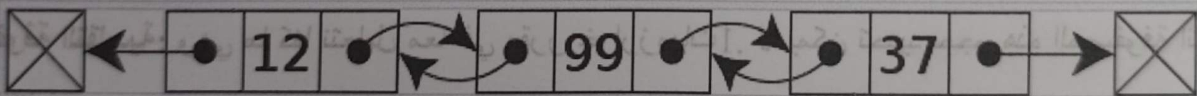
اللوائح المترابطة Linked List: هي نمط معطيات تشبه المصفوفات إلا أنه لا يتم فيها تخزين العناصر بشكل تتابعي متتالي، فهي تعتمد على مفهوم المؤشرات للربط بين العناصر.

تنقسم اللائحة المترابطة إلى أنواع أشهرها: اللائحة المفردة singly، والمزدوجة Doubled.

مثال على لائحة مترابطة مفردة: يوضح الشكل التالي أن كل عنصر (عقدة) يحوي على حقل قيمة، وعلى حقل مؤشر يشير إلى العنصر التالي.



مثال على لائحة مترابطة مزدوجة: يوضح الشكل التالي أن كل عنصر (عقدة) يحوي على حقل قيمة، وعلى حقل مؤشر يشير إلى العنصر التالي، وعلى حقل مؤشر يشير إلى العنصر السابق.



الفرق بين الدليل index في المصفوفات والمؤشر iterator في اللوائح المترابطة الجاهزة:

يستخدم الدليل للتجوال على عناصر المصفوفات، بينما يستخدم النمط iterator للتجوال على عناصر اللوائح في المكتبة الجاهزة.

الفائدة من النمط الجاهز iterator هي لتسهيل التجوال على عناصر اللائحة المترابطة الجاهزة، فهو يجنبنا التصريح عن مؤشر على اللائحة وتحريك هذا المؤشر يدوياً.

مثال على طريقة عمل List بواسطة مكتبة جاهزة هي #include <list>

سنستخدم التوابع الجاهزة التالية:

begin(): تابع يضع مؤشر على أول عنصر في اللائحة المترابطة.

end(): تابع يضع مؤشر بعد آخر عنصر في اللائحة المترابطة.

push_front(): تابع يدخل قيمة في بداية اللائحة.

push_back(): تابع يدخل قيمة في نهاية اللائحة.

pop_front(): تابع يخرج قيمة من بداية اللائحة.

pop_back(): تابع يخرج قيمة من نهاية اللائحة.

reverse(): تابع يعكس ترتيب عناصر اللائحة.

sort(): تابع يرتب عناصر اللائحة بشكل تصاعدي.

```
#include <iostream>
#include <list>
#include <iterator>
using namespace std;

//function for printing the elements in a list
void showlist(list <int> g)
{
    list <int> :: iterator it;
    for(it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}

int main()
{
    list <int> gqlist1, gqlist2;

    for (int i = 0; i < 10; ++i)
    {
        gqlist1.push_back(i * 2);
        gqlist2.push_front(i * 3);
    }
    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);

    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);

    cout << "\ngqlist1.front() : " << gqlist1.front();
    cout << "\ngqlist1.back() : " << gqlist1.back();

    cout << "\ngqlist1.pop_front() : ";
```

```
gqlist1.pop_front();
showlist(gqlist1);

cout << "\ngqlist2.pop_back() : ";
gqlist2.pop_back();
showlist(gqlist2);

cout << "\ngqlist1.reverse() : ";
gqlist1.reverse();
showlist(gqlist1);

cout << "\ngqlist2.sort(): ";
gqlist2.sort();
showlist(gqlist2);
system ("pause");
return 0;
}
```

الخرج:

```
List 1 (gqlist1) is :    0    2    4    6
8    10    12    14    16    18

List 2 (gqlist2) is :    27    24    21    18
15    12    9    6    3    0

gqlist1.front() : 0
gqlist1.back() : 18
gqlist1.pop_front() :    2    4    6    8
10    12    14    16    18

gqlist2.pop_back() :    27    24    21    18
15    12    9    6    3

gqlist1.reverse() :    18    16    14    12
10    8    6    4    2

gqlist2.sort():    3    6    9    12
15    18    21    24    27
```

المزايا الموجودة في اللوائح المترابطة والغير موجودة في المصفوفات:

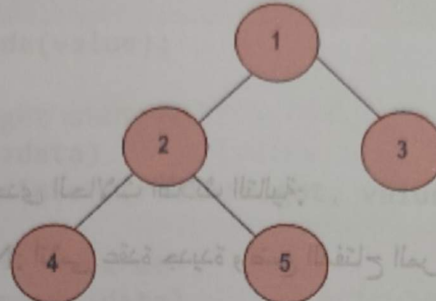
- سهولة حذف العناصر بسبب المؤشرات
- سهولة اضافة العناصر بسبب المؤشرات

المساوئ الموجودة في اللوائح المترابطة والغير موجودة في المصفوفات:

- عدم امكانية التجوال العشوائي والسريع، وانما يجب علينا الوصول تسلسلياً.
- تخصيص واستهلاك مساحات في الذاكرة لتخزين قيم المؤشرات.

التجوال في الأشجار الثنائية Binary Tree Traversals

ليكن لدينا الشجرة التالية:



يمكن تنفيذ التجوال بالعمق DFS على الأشجار الثنائية بثلاثة طرق:

1- التجوال الوسطي Inorder:

Inorder (Left, Root, Right) : 4 2 5 1 3

2- التجوال المسبق Preorder:

Preorder (Root, Left, Right) : 1 2 4 5 3

3- التجوال اللاحق Postorder:

Postorder (Left, Right, Root) : 4 5 2 3 1

بينما يمكن تنفيذ التجوال العرضي BFS عليها بطريقة واحدة:

Breadth First or Level Order Traversal : 1 2 3 4 5

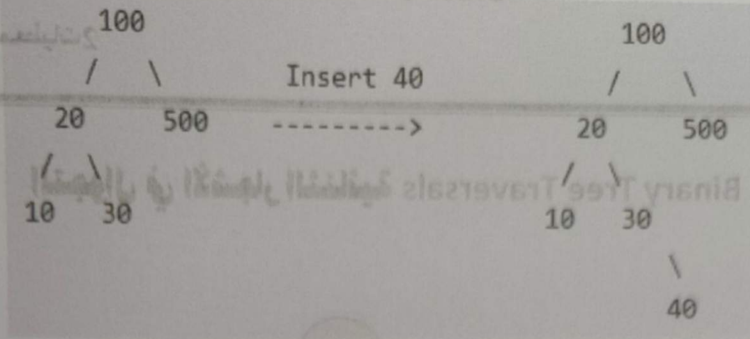
شجرة البحث الثنائية (Binary Search Tree)

هي الشجرة الثنائية التي تحقق التالي:

- في أي شجرة فرعية يسارية تكون قيم مفاتيح العقد أقل من قيمة مفتاح الجذر لها.
- في أي شجرة فرعية يمينية تكون قيم مفاتيح العقد أكبر من قيمة مفتاح الجذر لها.

إدخال قيمة Key في شجرة البحث الثنائية:

قيمة المفتاح الجديدة المراد إضافتها يجب دائماً أن تضاف كعقدة ورقة (Leaf) كما يوضح الشكل التالي عملية إضافة 40 كعقدة ورقة

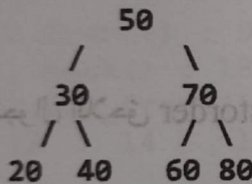


مبدأ عمل الخوارزمية:

عند ادخال عقدة جديدة فستكون لدينا إحدى الحالات الثلاث التالية:

- إذا كانت العقدة فارغة NULL: أنشئ عقدة جديدة وضع المفتاح المراد إدخاله فيها،
- أما إذا كانت قيمة المفتاح المراد إدخاله أكبر من مفتاح الجذر لها: استدعي الادخال لليمين،
- وإلا فقيمة المفتاح المراد إدخاله أصغر من مفتاح الجذر لها: استدعي الادخال لليسار.

مثال: اكتب كود خوارزمية إدخال (Insert) قيم شجرة البحث الثنائية التالية بلغة C++،



الحل:

ملاحظة: عند تطبيق التجوال inorder على شجرة بحث ثنائية فسيكون الخرج مرتب تصاعدياً، ويعتبر ذلك دليلاً على نجاح عمليات الإدخال في شجرة البحث الثنائي.

```
#include <iostream>
using namespace std;

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;

    // Constructor
    Node(int value) {
        data = value;
        left = right = NULL;
    }
};
```

```
// BST class (handles operations)
class BST {
public:

    // Insert a value into the BST
    Node* insert(Node* root, int value)
    {
        // If tree is empty, create new node
        if (root == NULL)
            return new Node(value);

        // Insert into right subtree
        if (value > root->data)
            root->right = insert(root->right, value);

        // Insert into left subtree
        else if (value < root->data)
            root->left = insert(root->left, value);

        return root;
    }

    // Inorder traversal (Left, Root, Right)
    void inorder(Node* root)
    {
        if (root == NULL)
            return;

        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
};

int main()
{
    BST tree;
    Node* root = NULL;

    // Insert values
    root = tree.insert(root, 50);
    tree.insert(root, 30);
    tree.insert(root, 20);
    tree.insert(root, 40);
    tree.insert(root, 70);
    tree.insert(root, 60);
    tree.insert(root, 80);

    // Print inorder traversal
```

```
    cout << "Inorder Traversal: ";
    tree.inorder(root);
     system("pause");
    return 0;
}
```

مثال: أضف تعديلاً في المكان المناسب بالأكواد السابقة يقوم بالتنبيه عن وجود إدخالات متكررة في شجرة البحث الثنائي، مع طباعة هذه الإدخالات

الحل: يجب مناقشة هذه الحالة في التابع insert قبل نهاية التابع كالتالي:

```
Node* insert(Node* root, int value)
{
    // If tree is empty, create new node
    if (root == NULL)
        return new Node(value);

    // Insert into right subtree
    if (value > root->data)
        root->right = insert(root->right, value);

    // Insert into left subtree
    else if (value < root->data)
        root->left = insert(root->left, value);

    // Handle duplicate values
    else
        cout << "Duplicate ignored: " << value << endl;

    return root;
}
```

مثال: أضف في المكان المناسب تابع يقوم بإيجاد عدد العقد في شجرة ثنائية، ثم استدعه في التابع main

الحل: ضمن الصف BST نكتب التابع count:

```
int count(Node* root)
{
    if (!root) return 0;
    return 1 + count(root->left) + count(root->right);
}
```

وتضاف تعليمة الاستدعاء التالية في التابع main كالتالي:

```
cout<<endl<<"The Count of Nodes is "<<tree.count(root);
```

مثال: أضف في المكان المناسب تابع يقوم يحسب ارتفاع شجرة ثنائية، ثم استدعه في التابع main

الحل: ضمن الصف BST نكتب التابع height:

```
int height(Node* root)
{
    if (!root) return 0;
    return 1 + max(height(root->left), height(root->right));
}
```

وتضاف تعليمة الاستدعاء التالية في التابع main كالتالي:

```
cout<<endl<<"The Height of Tree is "<<tree.height(root);
```